69665 U.S. PTO
12/30/96

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR UNITED STATES LETTERS PATENT

XPRESS MAIL CERTIFICATE OF MAILING 37 CFR 1.10
"Express Mail" mailing label number EM536403742US
Date of Deposit December 30, 1996
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.
Print Name VUDY L. STEINKRAUS
Signature
Date 12/30/96

FOR: **IMPROVED DITHERING METHOD AND APPARATUS USING RAMP PROBABILITY LOGIC**

INVENTOR: **DANIEL P. WILDE**

69665 U.S. PTO
12/30/96

IMPROVED DITHERING METHOD AND APPARATUS USING RAMP PROBABILITY LOGIC

72220 U.S. PTO
08777557

CROSS-REFERENCE TO RELATED APPLICATIONS

Not applicable.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

Not applicable.

BACKGROUND OF THE INVENTION

A. Field of the Invention

The present invention relates generally to a graphic system for a personal computer. More particularly, the present invention relates to rendering polygons on a computer screen. Still more particularly, the present invention relates to a technique for dithering polygons to create curved surfaces.

B. Background of the Invention

Before the availability of the personal computer (PC), computer graphics packages were expensive tools primarily reserved for industrial applications. Early microcomputers were only capable of rendering simple line drawings with a low screen resolution (256 x 256, for example). As microcomputers evolved, higher resolution color displays became available, and software applications routinely provided data output in a graphical format. The graphics techniques used were unstructured, with objects defined in terms of absolute coordinates using straight lines. Subsequently, graphics "primitives" were developed, enabling circles, ellipses, rectangles and polygons to be drawn with single software instructions. The use of primitives for drawing shapes

increased the speed at which the images can be rendered.

The availability of computer graphics has generated a demand for higher resolutions and three dimensional (3-D) rendering capabilities. Computer animation and games, in particular, have spawned a revolution in computer graphics capabilities. A 3-D image can be represented in a computer system as a collection of graphical objects, such as polygons, lines, and points. A set of vertex points defines a polygon. Associated with each point are certain parameters, such as shading, texturing, color, and the like. Identification of other non-vertex points within the polygon typically is done through the use of linear interpolation. Once interpolated, the polygon can be rendered on a computer monitor by successive scanning of successive rows of the polygon.

Demand for higher performance computer graphics has led to graphics systems capable of much greater speed, resolution, and quality than graphics systems of only a few years ago. Traditionally, graphic systems employed an "8-8-8" standard, and many still do, in which eight bits are used to represent each of the three primary colors red, green, and blue. That is, eight bits are used for red, eight bits for green, and eight bits for blue. In an 8-8-8 system, therefore, twenty-four bits are needed to fully represent the color of a single pixel. Using eight bits for each color allows 256 different shades of each of the three primary colors. To increase the resolution of the color system, each color is represented with an "8.16" interpolator value in which sixteen bits of fractional color are included with each eight bit integer shade. With such resolution 8-8-8 graphics systems typically produce very high quality color images.

The down side of an 8-8-8 graphics system is that twenty-four bits are necessary to represent the integer color shade of a single pixel. The number of pixels on a typical computer screen today commonly approaches 1,000,000. With twenty-four bits of color and other parameters

3

associated with each pixel, voluminous bits of information necessarily must be processed, thereby creating a demand for higher memory capacity and faster processing. Semiconductor part manufacturers have responded by developing higher performance graphics hardware. In many graphics applications, however, there is a demand for even higher performance from computer hardware.

Designers of graphics systems, therefore, have developed various techniques to achieve greater performance given the performance limitations of the supporting graphics hardware. For example, other graphics standards exist besides the 8-8-8 standard. One such standard is the "5-6-5" system in which five bits are used to represent shades of red, six bits are used to represent green, and five bits are used for blue. As shown in Figure 1, in an eight bit system, eight bits are used to represent the color red. In a 5-6-5 system, however, only the upper or most significant five bits, i.e., bits R_3-R_7, are used for red. The least significant three bits, R_0-R_2, are simply truncated and not used. For the color green the upper six bits are used, i.e., G_2-G_7, with bits G_0 and G_1 truncated. An extra bit is used for the color green because most human eyes are less sensitive to the color green and thus, additional resolution or precision for the color green is needed. The sixteen color bits that are used (five each for red and blue and six for green) are packed together for storage and/or processing. Other standards such as the "5-5-5" and "3-3-2" standards are also used.

Using a 5-6-5 system, instead of 8-8-8, advantageously allows the integer portions of the color representation to be stored in sixteen bits (i.e., two bytes) of memory instead of twenty-four bits (three bytes) as necessary for an 8-8-8 system. Moreover, significantly less memory is necessary to store computer images with a 5-6-5 system and more graphics information can be processed faster using existing hardware devices.

As one of ordinary skill in the art will recognize, representing a color shade with eight bits allows 2^8 or 256 different integer shades of that color. Representing a color shade with only five bits, however, allows for just 2^5 or 32 shades of that color and using six bits allows for 64 (2^6) color shades. Referring now to Figure 2, a comparison between an eight bit color standard and a five bit standard emphasizes the lower precision of a five bit system. Because the least significant three bits (bits 0-2) are truncated from an eight bit shade to create the five bit shade, eight bit color shades 0000 0000 through 0000 0111 (decimal 0-7) are represented by 00000 in the five bit system. In other words, with only five bits, the five bit representation cannot distinguish between eight bit shades 0-7 (decimal). Similarly, eight bit color shades 0000 1000 through 0000 1111 (decimal 8-15) are represented by 00001 in the five bit standard. This comparison illustrates that for every increment (or decrement) in a five bit color shade, eight eight-bit color shades are skipped. A similar comparison could be made for the six bit representation for the color green highlighting that because only the least significant two bits are truncated, there are four eight-bit green color shades for every six bit shade.

Polygons typically are drawn one row of pixels at a time, rendering pixels individually from one edge of the polygon to the other. To give the appearance of a curved surface to create 3-D images, a graphics system applies varying shades of color to the pixels rendered. For example, a polygon might be rendered with a dull shade of red on the left side of the polygon and bright red on the right side with a transition between the dull and bright shades for the intervening pixels. In an 8-8-8 system, with 256 integer shades each for red, green, and blue and sixteen bits of fractional shades, there is sufficient color precision to make the transitions of color across a polygon appear smooth to the human eye.

The attendant lower precision in a 5-6-5 graphics system does not permit color transitions across a polygon that are as smooth as in 8-8-8 systems. This problem is called "banding." Banding is caused by color transitions across polygons that include more pronounced, larger increments in shades of color because there are fewer different shades possible than in an 8-8-8 system. The boundary lines between different shades is more perceptible to the eye. A polygon rendered using the 5-6-5 color standard appears as bands of different shades of color on the computer screen rather than smooth transitions. Transitions in color shades in a 5-6-5 system between bands is noticeable despite the difference between adjacent bands of only a single shade of color.

To minimize the undesirable appearance of an image suffering from banding, the technique of dithering is used. Dithering takes advantage of the insufficient resolving ability of the human eye to distinguish individual pixels from a large group of pixels on a computer screen. Dithering allows the appearance of a color band to be altered to make the color shade of the band appear closer to the shade of an adjacent band, thereby smoothing the sharp transitions between the two bands. Graphics systems implementing known dithering techniques increment by one binary value randomly selected pixels in a band. Thus, some pixels in the dithered band are rendered using one shade of color, while other randomly selected pixels in the band are rendered using the color shade of the pixels in the adjacent band. The appearance of the dithered band thus appears closer to the shade of the adjacent band and the transition in color shades between the two bands is less noticeable to the eye. For the colors red and blue in a 5-6-5 system, incrementing a color shade by one shade level is equivalent to an increment of eight color shades in an eight bit system

as demonstrated in Figure 2. For the color green, a unitary increment equates to an increment of four shades of green in an eight bit system.

Although images using the dithering technique appear to lose the undesirable sharp edges between color bands, the result is still less than completely satisfactory because the dithered image is not an accurate rendition of the desired color transitions across the polygon. Because pixels are rendered randomly using one of two shades of color, the appearance to the eye of the band changes, but the resulting appearance is not necessarily the desired shade. Referring to Figure 2, for example, if the desired eight bit red color shade in a particular region of a polygon is 0000 0010, commonly known dithering techniques will randomly render the pixels in that region with five bit shades 00000 and 00001. If, however, eight bit shade 0000 0111 were desired, systems implementing dithering will also randomly render the same group of pixels with five bit shades 00000 and 00001. The appearance to the eye will be the same because the same five bit shades (00000 and 00001) would be used in random fashion in both instances.

Thus, it would be desirable to provide a computer graphics system that solves the banding problem associated with graphics systems that represent color with fewer than eight bits of precision. In particular, it would be desirable to provide a graphics system in which fewer than eight bits are used to represent shades of color that can achieve the color precision of eight bit systems. Despite the advantages of such a system, to date, no such system has been developed.

BRIEF SUMMARY OF THE INVENTION

A method and apparatus is disclosed for an improved technique to create the appearance of curved surfaces in a graphics system. The appearance of a curved surface is created by varying color shades from one edge of the surface to the other edge. To create high quality

curved surfaces, the color shade must be varied smoothly. The present invention preferably includes fewer than eight bits to represent shades of color. For example, five bits may be used for the colors red and blue and six bits may used for green. For sake of clarity, the disclosure of the invention assumes five bit color shade values. The five bit color shade values are derived from eight bit shade values by truncating the lower three bits of the eight bit shade values. Five bit color shade representations, although requiring less memory to store the shade values, provide less precision than eight bit shade values. Because fewer shades of color are possible with five bit systems, compared to eight bit systems, color "bands" are noticeable on the computer screen and detract from the appearance of the graphics images. The present invention overcomes the problem of lower precision by blending five bit shade values appropriately to create the appearance of a color shade representable otherwise only by an eight bit color shade value.

The appropriate blend of five bit color shade values is determined from the three bits that are truncated from the eight bit shade values to create the five bit shade values. The three truncated bits are referred to as the FRAC. The FRAC provides an indication of the color resolution that is lost when the FRAC bits are truncated to create the five bit shade values. The present invention allows for control of the blend of five bit color shade values to create the appearance of an eight bit color shade that could otherwise only be displayed in a graphics system that uses eight bits to represent color. The FRAC is used to control the blend of five bit color shade values. Thus, smooth color transitions can be made across a surface to create high quality curved surfaces and avoid banding problems associated with five and six bit systems.

BRIEF DESCRIPTION OF THE DRAWINGS

For a detailed description of the preferred embodiments of the invention, reference will now be made to the accompanying drawings wherein:

Figure 1 shows truncation of integer bits in a 5-6-5 graphics system;

Figure 2 depicts the reduction in resolution which results in a five bit system as compared to an eight bit system;

Figure 3 shows a block diagram of the dither system constructed in accordance with the present invention;

Figure 4 shows a look-up table constructed in accordance with the preferred embodiment;

Figure 5 shows a ramp table for the colors red and blue constructed in accordance with the preferred embodiment;

Figure 6 shows a ramp table for the color green constructed in accordance with the preferred embodiment; and

Figure 7 is a table illustrating the benefit of using the truncated bits to select the ramp value.

DETAILED DESCRIPTION OF THE INVENTION

Referring now to Figure 3, a dither system 10 constructed in accordance with the preferred embodiment is shown for dithering color in a five bit color system. It is recognized, however, that the invention could easily be adapted to a system using a different number of bits for representing color known by those of ordinary skill in the art upon reading the following disclosure. Thus, the present invention is intended to be operable with 5-6-5 and 3-3-2 color standards and the like.

Figure 3 is directed to a portion of a dither system for controlling dithering of the color red. One skilled in the art will understand that similar portions are used for the colors green and blue modified as needed depending on the number of bits used to represent green and blue color shades. Dither system 10 preferably includes a graphics processor 15, a look-up table 20, red addend generator 40, select fractional logic 50, dither probability logic 58, add logic 80, AND gate 90, and multiplexer 100. The graphics processor 15 includes commonly known processors such as the CL-GD5462 Visual Media ™ Accelerator (manufactured by Cirrus Logic), and the like. Although shown as a physically separate component in Figure 3, graphics processor 15 may include some or all of the other components shown. Graphics processor 15 preferably couples to a central processing core (not shown) and controls the rendering of images provided by the central processing core. The graphics processor 15 provides x and y addresses to the look-up table 20. Graphics controller 15 also provides a Red Interpolator signal on lines 30 to select fractional logic 50, add logic 80, and multiplexer 100. In addition, the graphics processor 15 provides a control signal to red addend generator 40.

Dither probability logic 58 includes a ramp generator 60 coupled to a multiplexer 70 via lines 65. Select fractional logic 50 provides input signals to ramp generator 60 over data lines 55. Look-up table 20 provides control signals to multiplexer 70 over data lines 25. As shown in Figure 3 and explained in greater detail below, multiplexer 70 is an 8:1 multiplexer in which one of eight input signals is selected to be the output signal in accordance with the state of the control signals provided by look-up table 20.

The inputs to the add logic 80 and select fractional logic 50 include a red interpolator signal on lines 30 from graphics processor 15. Inputs to add logic 80 also include the output of red

10

addend generator 40 over lines 45. The output of the add logic 80 couples to the "1" input of multiplexer 100 on lines 82. An overflow output signal (OVFLW) from add logic 80 on line 87 and the output signal of multiplexer 70 on line 72 are provided as input signals to AND gate 90. The input of AND gate 90 that receives the OVFLW signal preferably is an inverting input, but may be non-inverting depending on the active state of the OVFLW signal as described below. The red interpolator signal also is provided to the "0" input of multiplexer 100. The red interpolator signal preferably follows the 8.16 format. The output of the dither system 10 preferably is provided as the dithered red signal on line 105 which is the output of multiplexer 100.

The dither system 10 in Figure 3 illustrates generating a dithered signal for the color red, but similar architecture is used for dithering the colors blue and green. If a 5-6-5 color standard is implemented, the architecture for blue is substantially identical to that shown in Figure 3 because five bits are used for both red and blue in the 5-6-5 standard. Because six bits are used for green in the 5-6-5 format, some architectural differences result for dithering green. The present invention can be readily adapted to any color standard, as would be known by one ordinary skill in the art. Specific differences between the five bit dither system depicted in the drawings herein and six bit systems are identified throughout the following discussion as examples of how the invention can be adapted to graphics systems that represent color shades with more or less than five bits.

Referring now to Figure 4, look-up table 20 preferably comprises an 8 x 8 array of three bit numbers that are used for dithering the colors red and blue. The table entries are in the range of 000 to 111. For convenience the entries in the look-up table are shown in decimal form with the understanding that three bits are used to represent those values in binary form. The 64 three-bit values in the look-up table 20 may include many different combinations of three bit values, but the

combination shown in Figure 4 is preferred. Each pixel on the screen is identified by an x address and a y address preferably provided by graphics processor 15 (Fig. 3). Inputs to the look-up table 20 include the x and y addresses of the pixel to be rendered (i.e., colored). Because there are only eight columns and eight rows in the look-up table, only three bits are needed from the x address and y address to access all of the columns and rows, respectively. Preferably, only the least significant three bits of the x and y addresses are used to access the look-up table, although other combinations of three bits from the addresses could be used. Thus, if the least significant three bits of the x address are "101" (decimal 5) and the least significant three bits of the y address are "010" (decimal 2), the selected look-up table value will be "5." Look-up table 20 preferably is implemented in some type of random access memory (RAM) or hardware configuration registers, as will be apparent to one of ordinary skill in the art. For the color green, the look-up table 20 shown in Figure 4 may be used. Alternatively, look-up tables with only two bit values may be used.

Referring again to Figure 3, red addend generator 40 includes logic circuitry known to those of ordinary skill in the art for selecting an appropriate addend value which when added to a current eight bit shade of color results in an increment of color shade in a five bit representation following the truncation of the lower three bits. As can be seen with reference to Figure 1, to increment one five bit color shade for red and blue, a binary value of 1000 (decimal 8) must be added to an eight bit shade. Similarly, incrementing from one six bit green shade to the next higher shade requires adding a binary 100 (decimal) value to the eight bit shade because only the least significant two bits are truncated for green in the 5-6-5 system. After the appropriate addend value is added to the eight bit red shade represented by the Red Interpolator signal, the lower three bits are truncated with the resulting five bit red color shade representation being one five bit shade higher than it would have

been had the truncation occurred without the addition of the addend value. The addend generator 40 generates the appropriate addend value on lines 45 and provides that value to add logic 80. The addend generator 40 in Figure 3 is shown as for the color red. Additional addend generators can be provided for green and blue. Alternatively, one addend generator could be implemented that provides addend values for all three colors. The code in the Appendix includes exemplary logic equations for generating the addend values.

Add logic 80 adds the eight bit red integer interpolator value provided by processor 15 to the addend value received from red addend generator 40 and provides the sum to multiplexer 100 on output lines 82. The output lines 82 preferably include the upper five bits of the eight bit output value of add logic 80. By selecting only the upper five bits, the lower three bits are truncated. To dither the color green, the upper six bits of the add logic 80 output signal are used, thus truncating the least significant two bits.

Add logic 80 also preferably includes an overflow signal (OVFLW) on line 87. The OVFLW signal is a single bit value that indicates the existence of an overflow condition when adding binary values, the result of which requires an extra bit. Two eight bit values may be added, for example, and the result is a nine bit value. The additional ninth bit is referred to as the overflow bit. The OVFLW bit typically is a logic "1" to indicate an overflow condition or a "0" to indicate the absence of an overflow condition. Alternatively, a "0" might be used to indicate an overflow and a "1" might indicate the absence of an overflow. If the add logic 80 provides an OVFLW bit with the alternative protocol, the inverting input to AND gate 90 that receives the OVLFW bit should be replaced with a non-inverting input.

Select fractional logical 50 receives the red interpolator signal on line 30 from the graphics processor 15 and produces on its output lines 55 the three least significant bits of the eight bit integer shade (Figure 1). For dithering the color green, the two least significant bits are included on lines 55 by the select fractional logic 50. The values selected and output by the select fractional logic 50 are referred to as the fractional value or simply FRAC.

Ramp generator 60 receives the FRAC values from the select fractional logic 50 and includes logic for generating a multi-bit output value based upon the value of the FRAC. The multi-bit output value is selected from a ramp table. Referring now to Figure 5, an exemplary ramp table 61 comprises eight eight-bit ramp values. Table 61 preferably is used for dithering the colors red and blue in a 5-6-5 system. An eight bit ramp value is associated with each three bit FRAC value. As shown, a binary ramp value of 0000 0000 is associated with FRAC value 0, a ramp value 1000 0000 is associated with FRAC value 1, and a ramp value 1100 0000 is associated with FRAC value 2. Additionally, ramp value 1110 0000 is associated with FRAC value 3, ramp value 1111 0000 is associated with FRAC value 4, and ramp value 1111 1000 is associated with FRAC value 5. Finally, a ramp value of 1111 1100 is associated with FRAC value 6 and a ramp value of 1111 1110 is associated with FRAC value 7. The term "ramp" reflects the upward sloping appearance of table 61 indicated by line 62 separating the binary 1 values from the binary 0 values.

Referring to Figure 6, a ramp table 64 is shown for use in systems that use six bits to represent color. Table 64 is shown comprising four ramp values associated with four FRAC values. Each ramp value for the color green comprises a four bit binary value. As shown, ramp value 0000 is associated with FRAC value 0, and ramp value 1000 is associated with FRAC value 1. Finally,

14

ramp value 1100 is associated with FRAC value 2 and ramp value 1110 is associated with FRAC value 3.

The ramp tables 61, 64 shown in Figures 5 and 6 reflect the preferred embodiment of the invention. It should be noted, however, that ramp tables with different binary number combinations are possible and are also consistent with the preferred embodiment. Such other tables will become apparent to one of ordinary skill in the art upon reading this disclosure and thus are not shown explicitly herein.

As mentioned previously, multiplexer 70 preferably includes a commonly known 8:1 multiplexer including eight input signals on lines 65 and one output signal on line 72. One of the eight input signals from ramp generator 60 is selected by the multiplexer to be the output signal. The three bits on lines 25 generated by the look-up table 20 are used as control lines for multiplexer 70. The control bits determine which input signals the multiplexer should select. The multiplexer 70 decodes the control bits and switches or latches the input specified by the control bits to the output line. For example, if the bits on lines 25 include a 101 binary value (decimal 5), the fifth bit of the eight bit ramp value provided on lines 65 is selected and provided as the output signal on line 72. Thus, for a FRAC value of 6 and control bits on lines 25 of decimal 5, the bit circled in Figure 5 ("1") would be selected by multiplexer 70.

For six bit color systems, the ramp values preferably include four bits as discussed above with reference to Figure 6. In a six bit system, the multiplexer 70 comprises a 4:1 multiplexer. Because a 4:1 multiplexer includes four input signals, only two control bits are needed from the look-up table as would be apparent to one skilled in the art. Two of the three output bits from look-

up table 20 may be used (for example, the lower three bits) or the look-up table may be configured to include only two bit values as discussed previously.

Multiplexer 100 preferably includes two sets of input terminals, labeled "0" and "1" in Figure 3. Each set of input terminals includes five terminals. If a multiplexer is used that includes more than five input pins for each set of inputs, preferably only five of the terminals are implemented in the design for the purpose of dithering. The control signal for multiplexer 100 is provided from the output of AND gate 90 via line 92. A logic "0" control signal preferably directs the multiplexer 100 to select the "0" set of input lines and a logic "1" selects the "1" set of input lines. Alternatively, a "1" control bit might be used to select the "0" inputs and a "0" control bit might select the "1" inputs. The following discussion assumes the former protocol, that is, logic 0 control bit selecting "0" inputs and logic 1 control bit selecting "1" inputs. Whichever set of input bits ("0" or "1") is selected by the control bit, those selected bits are provided as the output signals of the multiplexer 100 on lines 105. The output bits on lines 105 represent the dithered red signal and are used to direct the operation of the red gun common to video monitors to render the appropriate dithered shade of red. The physical operation of the color guns in a cathode ray tube display are known to those of ordinary skill in the art and thus are not specifically discussed in this disclosure.

For the color green in which six bits are used to represent the integer shade of green, multiplexer 100 includes two sets of input signals with each set containing six signals. The "0" inputs include the upper six bits of the eight bit integer green shade value. The "1" inputs include the upper six bits of the output of the add logic 80 as discussed previously.

The principle upon which the present invention is based is described with reference to Figure 7 which shows nine eight-bit color shade values from 0000 0000 to 0000 1000 (decimal 0-8). Eight bit color shades 0 and 8 are accurately representable in a five bit system. That is, eight bit shade 0 is exactly equivalent to five bit shade 0 and eight bit shade 8 is exactly equivalent to five bit shade 1. Exact matching of color shades in eight bit and five bit systems occurs for any eight bit shade in which the lower three bits include only zero values. Color shade values 0000 0001 through 0000 0111 (decimal 1-7) are not accurately representable in a five bit system because the lower three truncated bits in box 63 contain non-zero values. Truncating the least significant three bits from eight bit shade values 1 through 7 results in five bit shade value 00000 as shown in box 64. The conversion to a five bit shade results in a shade value (00000) that is not equivalent to any of eight bit shade values 1 through 7. The loss of accuracy results from the lower precision capability of a five bit versus an eight bit system.

The truncated bits 63, however, indicate the number of incremental shades between eight bit values that can be represented exactly by a five bit shade. For example, eight bit color shade value 0000 0010 is two eight-bit integer color shades away from eight bit shade value 0000 0000 (which is equivalent to five bit shade 00000). Similarly, color shade value 0000 0111 is seven shades away from shade value 0000 0000. The truncated bits in box 63 provide a measure of the proximity, in terms of numbers of color shades, between the desired shade (0000 0010 and 0000 0111 in the examples above) and the nearest lower eight bit shade that is equivalent to a five bit shade.

Referring to Figures 3 and 7, the three truncated bits in box 63 represent the FRAC values and are produced by the select fractional logic 50. The present invention takes advantage of the fact that the three truncated bits, the FRAC value, provide an indication of the proximity of the desired

eight bit color shade value to an eight bit shade value that is equivalent to a shade value in a five bit system. With the FRAC value, dither system 10 renders pixels in a given color band using an appropriate mix of the two five bit color shade values closest to the desired eight bit shade. The appropriate blend of the two closest five bit shade values is determined by the FRAC value. It has been experimentally shown that a group of pixels can be rendered with two color shades (some pixels in the group rendered with one shade and other pixels with the other shade) to produce what appears to the eye to be a different shade than either of the two shades used to render the pixels. The resulting apparent color shade, in fact, can be controlled by varying the mixture of the two shades; that is, controlling which pixels in the group are rendered with one shade and which pixels are rendered with the other shade.

With reference to Figure 7, if for example, it is desired to render a portion of an image with the eight bit red color shade value 0000 0111 (decimal 7) in a five bit system, the pixels can be colored with five bit shade values 00000 and 00001; some of the pixels with shade value 00000 and other pixels with shade value 00001. If the mixture of shade values 00000 and 00001 is determined appropriately, the image will appear as eight bit shade 0000 0111. The bits truncated during the conversion of the eight bit integer shade value to a five bit value (the FRAC value) are used for controlling which pixels are rendered with five bit shade value 00000 and which pixels are rendered with shade value 00001.

If the desired eight bit shade value is 0000 0111 (decimal 7), the corresponding FRAC value is 111 indicating that the desired eight bit shade value is seven color shades from five bit shade value 00000. There are eight eight-bit shade values for every five bit shade value as shown best by reference A in Figure 2. Thus, eight bit shade value 0000 0111 can be thought of as being

7/8 of the total number of eight bit shade values between 0000 0000 and 0000 1000. The appearance of eight bit shade value 0000 0111 in a group of pixels in a five bit system can be created by rendering 7/8 of all of the pixels in the group as five bit shade value 00001 and the remaining 1/8 of the pixels as five bit shade value 00000. The selection of pixels to be rendered as shade value 00000 or 00001 is not critical and preferably is substantially random, i.e., a randomly selected portion of the pixels in the group are rendered with shade value 00001. By way of a further example, eight bit shade value 0000 0010 has an associated FRAC of 010 (decimal 2) and thus, this eight bit shade value is 2/8 of the total number of eight bit color shade increments between shade values 0000 0000 and 0000 1000. Thus, 2/8 of the pixels in a group for which it is desired to appear as eight bit shade value 0000 0010 are rendered with five bit shade value 00001 and 6/8 of the pixels are rendered with shade value 00000. The left most column in the table in Figure 2 indicates the fractions associated with each eight bit shade that does not have an equivalent five bit shade.

Referring to Figures 5 and 6, the proximity information from the FRAC values is encoded in the ramp values in tables 61, 64 through the number of logic 1 values. Thus, ramp value 1111 1110, associated with FRAC value 7, includes seven logic 1 bits. Similarly, ramp value 1000 0000 includes one logic 1 and is associated with FRAC value 1.

It will be apparent to one of ordinary skill in the art that because the number of logic one values in the ramp tables encodes the desired proximity information, any one ramp value need only include the proper number of logic 1 values; it is not important which bit positions contain the logic 1 and 0 values. Thus, ramp value 1000 0000, for example can be substituted with 0100 0000, 0010 0000, 0001 0000, 0000 1000, 0000 0100, 0000 0010, and 0000 0001.

Referring to Figure 3, ramp generator 60 uses the FRAC value received on lines 55 to produce on its output lines 65 a corresponding eight bit ramp value per tables 61, 64. Multiplexer 70 receives the eight bit RAMP value from ramp generator 60 and the three bit value from look-up table 20. The three bit value from look-up table 20 is used to select one of the eight ramp bits on lines 65. The bit that is selected from the ramp value is provided on the multiplexer's output line 72. The probability that an output signal from multiplexer 70 will be a logic 1 depends on the number of logic 1's in the ramp value. If a 1111 1100 ramp value, for example, is provided to the multiplexer and one of those bits is randomly selected, the probability that the selected bit will be a logic 1 is 6/8 or 75% because six of the eight bits comprises a logic 1. However, if the ramp value was 1100 0000, the probability that the selected bit will be a logic 1 is 2/8 or 25%. As will be seen below, the probability that the output bit from multiplexer 70 is a logic 1 directly determines the mix of color shades for dithering.

As stated previously, the look-up table 20 provides the control bits to the multiplexer 70 and are used to select the output bit from the eight input ramp bits. The combination of three bit entries in look-up table 20 are not completely randomly selected values, but have been selected in accordance with the preferred embodiment because that combination has been shown experimentally to provide superior dither results to other three bit combinations.

Still referring to Figure 3, the addend value from red addend generator 40 is added to the red interpolator integer value by add logic 80 and the most significant five bits are provided on the add logic's output lines 82. The output of the add logic 80 thus includes the five bit color shade that is one five bit shade higher than the five bit shade resulting from truncating the least significant three bits of the eight bit shade. The "0" input to multiplexer 100 includes the five bit shade

70

resulting from truncating the lower three bits of the eight bit shade by add logic 80. The "1" input includes the shade on the "0" input incremented by one shade by add logic 80. Multiplexer 100 is used to select one of the two five bit shades for dithering. The output of AND gate 90 provides the control bit to select between the inputs of multiplexer 100.

With the OVFLW signal set to 0, indicating the absence of an overflow condition in add logic 80, the state of the output bit from multiplexer 70 dictates the state of the control line for multiplexer 100. If the output bit from multiplexer 70 is a logic 0, the output of AND gate 90 will be a logic 0 and the "0" input lines of multiplexer 100 will be selected for the output on lines 105. Conversely, if the output bit from multiplexer 70 is a logic 1, the output of AND gate 90 will be a logic 1 and the "1" input lines from multiplexer 70 will be selected for the output on lines 105.

The logic level of the output bit of multiplexer 70 on line 72 will be the same as the logic level of the control bit for multiplexer 100 and thus, the selection of the five bit shade on input "0" or the five bit shade on input "1" (which is one shade level higher than the shade on input "0") is directed by the output bit from multiplexer 70. The probability that input "1" will be selected is the same as the probability that the output bit of multiplexer 70 will be a logic 1. It can thus be seen that the selection of inputs "0" and "1" follow the number of logic 1's in the ramp values. The resulting dithered red output signal on lines 105 provides the appropriate mix of five bit shades to create the appearance in a group of pixels of an eight bit shade that has no equivalent shade in a five bit system.

It is possible that the addition of the red interpolator value on lines 30 and the red addend generator 40 output on lines 45 creates an overflow condition as one of ordinary skill in the art will know. If the desired eight bit red color shade value is 1111 1111, for example, and 0001 0000

represents an appropriate red addend generator output value and is added to the desired eight bit color value by add logic 80, the result is 1 0000 1111. If only the upper five bits (not including the ninth overflow bit) of the output of add logic 80 are used, the resulting dithered red signal selected by multiplexer 100, with control line 92 at a logic 1 state, would be 00001. Generally, the lowest color shade value represents the dullest shade and the highest color shade value represents the brightest shade. Adding addend value 0001 0000 to eight bit color shade value 1111 1111 (bright red) to generate the next highest five bit shade creates, instead, five bit color shade 00001 (dull red). The OVFLW bit is used to avoid rendering an erroneous color shade.

The overflow bit from add logic 80 is input into an inverting input of AND gate 90. If the OVFLW bit is a 1 indicating the presence of an overflow condition, the output of AND gate 90 will be a zero and thus, multiplexer 100 input "0" will be selected. Thus, for overflow conditions, the sum of the red addend generator output value and the eight bit interpolator value will not be selected as the dithered red output signal on line 105. Instead, the upper five bits of the eight bit interpolator value on input "0" of multiplexer 100 will always be selected during overflow conditions. The dithering function effectively is disabled during overflow situations. Disabling dithering during overflows is preferable to changing bright color shades to dull color shades, and vice versa.

While a preferred embodiment of the invention has been shown and described, modifications thereof can be made by one skilled in the art without departing from the spirit of the invention.

```
-- filename ncolorsr1.vhd
library work;
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;


entity ncolor_src_1 is
port(   clk: in std_logic;
        pattern_reg_0: in std_logic_vector(31 downto 0);
        pattern_reg_1: in std_logic_vector(31 downto 0);
        pattern_reg_2: in std_logic_vector(31 downto 0);
        pattern_reg_3: in std_logic_vector(31 downto 0);
        pattern_reg_4: in std_logic_vector(31 downto 0);
        pattern_reg_5: in std_logic_vector(31 downto 0);
        pattern_reg_6: in std_logic_vector(31 downto 0);
        pattern_reg_7: in std_logic_vector(31 downto 0);
        pat_adrs: in std_logic_vector(2 downto 0);
        pat_rd_strb: in std_logic;
        pat_data_out: buffer std_logic_vector(31 downto 0);

        color_reg_0: in std_logic_vector(23 downto 0);
        color_reg_1: in std_logic_vector(23 downto 0);

        draw_line: in std_logic;
        stipple_enable: in std_logic;
        dither_enable: in std_logic;
        pixel_mode: in std_logic_vector(1 downto 0);
        color_mode_332: in std_logic;

        pat_poly: in std_logic;
        pat_y_offset: in std_logic_vector(3 downto 0);
        pat_x_offset: in std_logic_vector(3 downto 0);

        line_adrs: std_logic_vector(4 downto 0);
        itp_y_adrs: std_logic_vector(3 downto 0);
        itp_x_adrs: std_logic_vector(3 downto 0);
        itp_red: std_logic_vector(7 downto 0);
        itp_red_frac: std_logic_vector(2 downto 0);
        itp_green: std_logic_vector(7 downto 0);
        itp_green_frac: std_logic;
        itp_blue: std_logic_vector(7 downto 0);

        mask_out: buffer std_logic; -- 0=don't write pixel

        alpha: in std_logic;
        itp_alpha: in std_logic_vector(3 downto 0);

        red_out: buffer std_logic_vector(7 downto 0);
        green_out: buffer std_logic_vector(7 downto 0);
        blue_out: buffer std_logic_vector(7 downto 0));
end;


-- decode of configuration bits
--
-- stipple_enable
--      1 = enabled
--      0 = disabled
--
-- dither_enable
--      1 = enabled
--      0 = disabled
--
-- pat_poly
--      1 = color/pattern regs
--      0 = interpolators
--


architecture behavioral of ncolor_src_1 is

component patmux
port(   reg_sel: in std_logic_vector(2 downto 0);
        nib_sel: in std_logic_vector(2 downto 0);
        bit_sel: in std_logic_vector(1 downto 0);
        pattern_reg_0: in std_logic_vector(31 downto 0);
        pattern_reg_1: in std_logic_vector(31 downto 0);
```

```
        pattern_reg_2: in std_logic_vector(31 downto 0);
        pattern_reg_3: in std_logic_vector(31 downto 0);
        pattern_reg_4: in std_logic_vector(31 downto 0);
        pattern_reg_5: in std_logic_vector(31 downto 0);
        pattern_reg_6: in std_logic_vector(31 downto 0);
        pattern_reg_7: in std_logic_vector(31 downto 0);
        reg_out: buffer std_logic_vector(31 downto 0);
        pattern_out: buffer std_logic_vector(3 downto 0));
end component;

component pipereg1
port(   clk: in std_logic;
        d_in: in std_logic;
        q_out: buffer std_logic);
end component;

component pipereg3
port(   clk: in std_logic;
        d_in: in std_logic_vector(2 downto 0);
        q_out: buffer std_logic_vector(2 downto 0));
end component;

component pipereg4
port(   clk: in std_logic;
        d_in: in std_logic_vector(3 downto 0);
        q_out: buffer std_logic_vector(3 downto 0));
end component;

component pipereg8
port(   clk: in std_logic;
        d_in: in std_logic_vector(7 downto 0);
        q_out: buffer std_logic_vector(7 downto 0));
end component;


signal  pattern_out,pattern_out_p1: std_logic_vector(3 downto 0);

signal  red_plus_one: std_logic_vector(8 downto 0);
signal  green_plus_one: std_logic_vector(8 downto 0);
signal  blue_plus_one: std_logic_vector(8 downto 0);

signal  red_overflow: std_logic;
signal  green_overflow: std_logic;
signal  blue_overflow: std_logic;

signal  red_src_p1,green_src_p1,blue_src_p1: std_logic_vector(7 downto 0);

signal  itp_red_frac_sel_p1,itp_green_frac_sel_p1,itp_blue_frac_sel_p1: std_logic_vector(2 downto 0);
signal  red_frac_src_p1: std_logic_vector(2 downto 0);
signal  green_frac_src_p1: std_logic;
signal  red_dither_sel_p1,green_dither_sel_p1,blue_dither_sel_p1: std_logic;
signal  rand_dith_sel_p1: std_logic_vector(2 downto 0);

signal  pat_reg_sel,nib_sel: std_logic_vector(2 downto 0);
signal  bit_sel: std_logic_vector(1 downto 0);
signal  offset_itp_y_adrs,offset_itp_x_adrs: std_logic_vector(3 downto 0);
signal  red_dither_addend,green_dither_addend,blue_dither_addend: std_logic_vector(7 downto 0);
signal  pm_8bpp,pm_16bpp: std_logic;
signal  pat_0,pat_1: std_logic;
signal  red_plus,green_plus,blue_plus: std_logic;
signal  red_pass,green_pass,blue_pass: std_logic;
signal  red_ahead,green_ahead,blue_ahead: std_logic;

signal  tmp_lookup: std_logic_vector(2 downto 0);


signal  alpha_stipple_sel_p1: std_logic;
signal  itp_alpha_sel_p1: std_logic_vector(3 downto 0);

signal  pat_8_8: std_logic;

begin

-- constants

pm_8bpp  <= not pixel_mode(1) and not pixel_mode(0);
pm_16bpp <= not pixel_mode(1) and     pixel_mode(0);

red_dither_addend   <= "00" & (pm_8bpp and color_mode_332) & "0"  & pm_16bpp & "00" & (pm_8bpp and not color_mode_332);
green_dither_addend <= "00" & (pm_8bpp and color_mode_332) & '0' & (pm_16bpp and color_mode_332) & (pm_16bpp and not color_mc
```

```
blue_dither_addend  <= "0"  & (pm_8bpp and color_mode_332) & "00" & pm_16bpp & "000";

u_red_src_p1: pipereg8 port map(clk,itp_red_src_p1);
u_green_src_p1: pipereg8 port map(clk,itp_green,green_src_p1);
u_blue_src_p1: pipereg8 port map(clk,itp_blue,blue_src_p1);

u_red_frac_src_p1: pipereg3 port map(clk,itp_red_frac(2 downto 0),red_frac_src_p1);
u_green_frac_src_p1: pipereg1 port map(clk,itp_green_frac,green_frac_src_p1);



rand_dith_sel_p1 <= pattern_out_p1(2 downto 0);

itp_red_frac_sel_p1 <=  red_src_p1(2 downto 0) when (pm_16bpp='1') else
                        red_frac_src_p1(2 downto 0) when (color_mode_332='0') else
                        red_src_p1(4 downto 2);

itp_green_frac_sel_p1 <= (green_src_p1(1 downto 0) & green_frac_src_p1) when ((pm_16bpp='1') and (color_mode_332='0')) else
                         green_src_p1(2 downto 0) when (pm_16bpp='1') else
                         green_src_p1(4 downto 2);

itp_blue_frac_sel_p1 <= blue_src_p1(2 downto 0) when (pm_16bpp='1') else
                        blue_src_p1(5 downto 3);


dithselmux: process(itp_red_frac_sel_p1,itp_green_frac_sel_p1,itp_blue_frac_sel_p1,rand_dith_sel_p1,
                    red_dither_sel_p1,green_dither_sel_p1,blue_dither_sel_p1)
variable red_ramp,blue_ramp,green_ramp: std_logic_vector(6 downto 0);
begin

case itp_red_frac_sel_p1 is
        when "111" =>   red_ramp:="1111111";
        when "110" =>   red_ramp:="0111111";
        when "101" =>   red_ramp:="0011111";
        when "100" =>   red_ramp:="0001111";
        when "011" =>   red_ramp:="0000111";
        when "010" =>   red_ramp:="0000011";
        when "001" =>   red_ramp:="0000001";
        when others =>  red_ramp:="0000000";
end case;

case itp_green_frac_sel_p1 is
        when "111" =>   green_ramp:="1111111";
        when "110" =>   green_ramp:="0111111";
        when "101" =>   green_ramp:="0011111";
        when "100" =>   green_ramp:="0001111";
        when "011" =>   green_ramp:="0000111";
        when "010" =>   green_ramp:="0000011";
        when "001" =>   green_ramp:="0000001";
        when others =>  green_ramp:="0000000";
end case;

case itp_blue_frac_sel_p1 is
        when "111" =>   blue_ramp:="1111111";
        when "110" =>   blue_ramp:="0111111";
        when "101" =>   blue_ramp:="0011111";
        when "100" =>   blue_ramp:="0001111";
        when "011" =>   blue_ramp:="0000111";
        when "010" =>   blue_ramp:="0000011";
        when "001" =>   blue_ramp:="0000001";
        when others =>  blue_ramp:="0000000";
end case;


case rand_dith_sel_p1 is
        when "000" =>
                        red_dither_sel_p1 <= red_ramp(0);
                        green_dither_sel_p1 <= green_ramp(0);
                        blue_dither_sel_p1 <= blue_ramp(0);
        when "001" =>
                        red_dither_sel_p1 <= red_ramp(1);
                        green_dither_sel_p1 <= green_ramp(1);
                        blue_dither_sel_p1 <= blue_ramp(1);
        when "010" =>
                        red_dither_sel_p1 <= red_ramp(2);
                        green_dither_sel_p1 <= green_ramp(2);
                        blue_dither_sel_p1 <= blue_ramp(2);
        when "011" =>
                        red_dither_sel_p1 <= red_ramp(3);
```

```
                    green_dither_sel_p1 <= green_ramp(3);
                    blue_dither_sel_p1 <= blue_ramp(3);
        when "100" =>
                    red_dither_sel_p1 <= red_ramp(4);
                    green_dither_sel_p1 <= green_ramp(4);
                    blue_dither_sel_p1 <= blue_ramp(4);
        when "101" =>
                    red_dither_sel_p1 <= red_ramp(5);
                    green_dither_sel_p1 <= green_ramp(5);
                    blue_dither_sel_p1 <= blue_ramp(5);
        when "110" =>
                    red_dither_sel_p1 <= red_ramp(6);
                    green_dither_sel_p1 <= green_ramp(6);
                    blue_dither_sel_p1 <= blue_ramp(6);
        when others =>
                    red_dither_sel_p1 <= '0';
                    green_dither_sel_p1 <= '0';
                    blue_dither_sel_p1 <= '0';
end case;


end process dithselmux;


offset_itp_y_adrs <= itp_y_adrs + pat_y_offset;

offset_itp_x_adrs <= itp_x_adrs + pat_x_offset;

pat_8_8 <= (dither_enable or (alpha and stipple_enable));


pat_reg_sel <= pat_adrs when (pat_rd_strb='1') else
               offset_itp_y_adrs(2 downto 0) when (pat_8_8='1') else
               pat_y_offset(2 downto 0) when (draw_line='1') else
               offset_itp_y_adrs(3 downto 1);

nib_sel     <=  offset_itp_x_adrs(2 downto 0) when (pat_8_8='1') else
                line_adrs(4 downto 2)  when (draw_line='1') else
                (offset_itp_y_adrs(0) & offset_itp_x_adrs(3 downto 2));


bit_sel     <=  "11" when (pat_8_8='1') else
                line_adrs(1 downto 0) when (draw_line='1') else
                offset_itp_x_adrs(1 downto 0);

u_patmux: patmux port map(
        pat_reg_sel,
        nib_sel,
        bit_sel,
        pattern_reg_0,
        pattern_reg_1,
        pattern_reg_2,
        pattern_reg_3,
        pattern_reg_4,
        pattern_reg_5,
        pattern_reg_6,
        pattern_reg_7,
        pat_data_out,
        pattern_out);


u_pat_out_p1: pipereg4 port map(clk,pattern_out,pattern_out_p1);



red_plus_one    <= ('0' & red_src_p1)   + red_dither_addend;
green_plus_one  <= ('0' & green_src_p1) + green_dither_addend;
blue_plus_one   <= ('0' & blue_src_p1)  + blue_dither_addend;


red_ahead <= red_dither_sel_p1 and not red_plus_one(8);
red_pass <=not pat_poly and (not dither_enable or (dither_enable and not red_ahead));
red_plus <= dither_enable and not pat_poly and red_ahead;

green_ahead <= green_dither_sel_p1 and not green_plus_one(8);
green_pass <=not pat_poly and (not dither_enable or (dither_enable and not green_ahead));
green_plus <= dither_enable and not pat_poly and green_ahead;

blue_ahead <= blue_dither_sel_p1 and not blue_plus_one(8);
blue_pass <=not pat_poly and (not dither_enable or (dither_enable and not blue_ahead));
```

```
blue_plus <= dither_enable and not pat_poly and blue_ahead;


pat_0 <= pat_poly and not pattern_out_p1(3);
pat_1 <= pat_poly and     pattern_out_p1(3);

outmuxred: for i in 0 to 7 generate
        red_out(i) <=   (red_src_p1(i)   and red_pass) or
                        (red_plus_one(i) and red_plus) or
                        (color_reg_0(i+16)  and pat_0) or
                        (color_reg_1(i+16)  and pat_1);
end generate;

outmuxgreen: for i in 0 to 7 generate
        green_out(i) <=   (green_src_p1(i)       and green_pass) or
                        (green_plus_one(i) and green_plus) or
                        (color_reg_0(i+8)  and pat_0) or
                        (color_reg_1(i+8)  and pat_1);
end generate;

outmuxblue: for i in 0 to 7 generate
        blue_out(i) <=   (blue_src_p1(i)         and blue_pass) or
                        (blue_plus_one(i) and blue_plus) or
                        (color_reg_0(i)  and pat_0) or
                        (color_reg_1(i)  and pat_1);
end generate;



--mask_out <= not stipple_enable or pattern_out_p1(3);

mask_out <= (not stipple_enable or ((not alpha and pattern_out_p1(3)) or
                                    (alpha and alpha_stipple_sel_p1)));


u_alpha_src_p1: pipereg4 port map(clk,itp_alpha,itp_alpha_sel_p1);

alphaselmux: process(itp_alpha_sel_p1,pattern_out_p1,
                     alpha_stipple_sel_p1)
variable alpha_ramp: std_logic_vector(14 downto 0);
begin

case itp_alpha_sel_p1 is
        when "1111" =>  alpha_ramp:="111111111111111";
        when "1110" =>  alpha_ramp:="011111111111111";
        when "1101" =>  alpha_ramp:="001111111111111";
        when "1100" =>  alpha_ramp:="000111111111111";
        when "1011" =>  alpha_ramp:="000011111111111";
        when "1010" =>  alpha_ramp:="000001111111111";
        when "1001" =>  alpha_ramp:="000000111111111";
        when "1000" =>  alpha_ramp:="000000011111111";
        when "0111" =>  alpha_ramp:="000000001111111";
        when "0110" =>  alpha_ramp:="000000000111111";
        when "0101" =>  alpha_ramp:="000000000011111";
        when "0100" =>  alpha_ramp:="000000000001111";
        when "0011" =>  alpha_ramp:="000000000000111";
        when "0010" =>  alpha_ramp:="000000000000011";
        when "0001" =>  alpha_ramp:="000000000000001";
        when others =>  alpha_ramp:="000000000000000";
end case;


case pattern_out_p1 is
        when "0000" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(0);
        when "0001" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(1);
        when "0010" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(2);
        when "0011" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(3);
        when "0100" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(4);
        when "0101" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(5);
        when "0110" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(6);
```

```
        when "0111" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(7);
        when "1000" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(8);
        when "1001" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(9);
        when "1010" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(10);
        when "1011" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(11);
        when "1100" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(12);
        when "1101" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(13);
        when "1110" =>
                        alpha_stipple_sel_p1 <= alpha_ramp(14);
        when others =>
                        alpha_stipple_sel_p1 <= '0';

end case;

end process alphaselmux;




end;
```

Page 23 Removed
for CRC.
6 pages Removed.
1-6/2